Eurographics Symposium on Geometry Processing (2005)
M. Desbrun, H. Pottmann (Editors)

Streaming Compression of Triangle Meshes

Martin Isenburg[1†] Peter Lindstrom[2] Jack Snoeyink[1]

[1] University of North Carolina at Chapel Hill [2] Lawrence Livermore National Labs

Abstract

Current mesh compression schemes encode triangles and vertices in an order derived from systematically traversing the connectivity graph. These schemes struggle with gigabyte-sized mesh input where the construction and the usage of the data structures that support topological traversal queries become I/O-inefficient and require large amounts of temporary disk space. Furthermore they expect the entire mesh as input. Since meshes cannot be compressed until their generation is complete, they have to be stored at least once in uncompressed form.

We radically depart from the traditional approach to mesh compression and propose a scheme that incrementally encodes a mesh in the order it is given to the compressor using only minimal memory resources. This makes the compression process essentially transparent to the user and practically independent of the mesh size. This is especially beneficial for compressing large meshes, where previous approaches spend significant memory, disk, and I/O resources on pre-processing, whereas our scheme starts compressing after receiving the first few triangles.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Boundary representations

1. Introduction

Modern technology enables the creation of three-dimensional polygonal models with incredible precision and the resulting data sets easily reach file sizes of several gigabytes. The St. Matthew model from Stanford's Digital Michelangelo Project [LPC*00], for example, is more than six gigabytes of data if stored in a standard indexed mesh format. The need for more compact mesh representations has motivated research on mesh compression and since 1995 many efficient schemes have been proposed [Dee95, TG98, GS98, Ros99, LAD02, KPRW05].

However, current mesh compression schemes are not particularly suited to deal with gigabyte-sized meshes. All these schemes struggle for the same three reasons with large input meshes: First, they expect the *entire* uncompressed mesh to be available as input. This means that a newly generated mesh needs to be stored at least once in uncompressed form (i.e. in order to hand it to the compressor) and that compression cannot start until mesh generation has completed. Second, prior to compression they need to construct temporary data structures in the size of the mesh that support topological adjacency queries. For gigabyte-sized input, the construction of such structures requires enormous amounts of memory resources. Third, they completely reorder the input mesh based on a deterministic traversal of the connectivity graph. For large meshes, this requires reorganizing gigabytes of data on a global scale—an overall expensive operation.

Most compression schemes can simply not be used for large meshes as constructing the required data structures is not possible given the limited memory resources on common PCs. Addressing this issue, Ho et. al [HLK01] suggest cutting large meshes into manageable pieces, encoding each separately using previous techniques, and recording additional information that specifies how to stitch the pieces back together. Instead, to avoid the cutting step and compress gigantic meshes in one piece, Isenburg and Gumhold [IG03] propose a dedicated external memory data structure that supports the topological adjacency queries of their compressor. However, both these approaches are highly I/O-inefficient and require large amounts of temporary disk space.

We radically depart from the traditional approach to mesh compression and propose a *streaming compression* scheme that incrementally encodes a mesh in the order it is given to the compressor using only minimal memory resources. This makes the compression process basically transparent to the user and practically independent of the mesh size. This is especially beneficial for compressing large meshes where all previous approaches need to spend significant amounts of main memory, temporary disk space, and file I/O on preprocessing the input mesh, whereas our scheme can start compressing after having been given the first few triangles.

† isenburg@cs.unc.edu http://www.cs.unc.edu/ ~ isenburg/smc

We have implemented a *writer* (see Figure 1) and a corresponding *reader* through which compressed meshes can be written and read in increments of single vertices and triangles. The only requirement is that vertices are written only shortly before being referenced by a triangle and that vertices are "finalized" with the triangle that references them for the last time (e.g. by a boolean flag). Outputting vertices and triangles in an interleaved fashion while providing explicit "finalization" information means that the mesh is written in a streaming format, or that it is a *streaming mesh* [IL05].

The required changes for writing meshes to a streaming instead of to a standard indexed format are minimal. Most mesh generating applications that operate with limited memory naturally output streaming meshes. For example, an out-of-core implementation of iso-surface extraction from regular grids that processes the volume layer by layer will naturally produce vertices and triangles in an interleaved fashion and can trivially finalize vertices from the last layer before moving on to the next. For such applications it is, in fact, difficult to output meshes into standard noninterleaved formats.

One fundamental disadvantage of streaming connectivity compression is that there are no guarantees on the bit-rates. The achieved compression strongly depends on the order in which the triangles are compressed and incoherent orderings give poor results. When our compressor follows the exact order in which the triangles are written, it generally needs to store at least $\log_2(\mathit{width})$ bits per triangle, where *width* is the number of previously referenced, yet unfinalized vertices. However, since all previous compression schemes are allowed to globally reorder the mesh triangles, it seems only fair to allow our compressor at least some local reordering.

We can significantly improve compression rates by employing a small *delay buffer* within which the compressor can locally reorder the triangles. It greedily brings them into a vertex-connected order that often allows avoiding those $\log_2(\mathit{width})$ bits. A delay buffer of 10,000 triangles, for example, does not significantly increase the overall memory requirements. But it leads to connectivity rates of around 4 to 5 bits per vertex for reasonably coherent input meshes, which is within a factor of two of previous schemes.

2. Preliminaries and Related Work

Standard indexed mesh formats store an array of floats triplets to specify the vertex positions (i.e. the geometry) and an array of integers triplets that index into the vertex array to specify the triangles (i.e. the connectivity). In this format, the cost for storing connectivity increases super-linearly in the number of vertices v as each index requires at least $\log_2(v)$ bits. The overall storage costs are about $6\log_2(v)$ bits per vertex as each vertex is indexed an average of six times.

An indexed format not only specifies the connectivity but also the particular order in which vertices and triangle appear in their respective array. A mesh with v vertices and t triangles can be listed in $v! \cdot t! \cdot 3^t$ ways by chosing any permutation of all vertices and of all triangles, and then chosing the rotation of each triangle. The log-factor in the storage cost comes from specifying one of these many orderings.

```
typedef Type enum {SM_VERTEX, SM_TRIANGLE, SM_EOF};
class SMwriter_smc {
  // may optionally be set if known in advance
  void set_bounding_box(float* min, float* max);
  void set_num_verts(int nverts);
  void set_num_faces(int nfaces);
  // finalize vertices used for the last time
  bool open(FILE* file, int bits = 16);
  bool write_vertex(float* v_pos);
  bool write_triangle(int* t_idx, bool* t_final);
  bool close();
}
class SMreader_smc {
  int bits;
  float *bb_min, *bb_max;  // only optionally known
  int nverts, nfaces;      // only optionally known
  bool open(FILE* file);
  Type read_element();
  bool close();
  float* v_pos;            // position of read vertex
  int* t_idx;              // indices of read triangle ...
  bool* t_final;           // ... and their finalization
}
```

Figure 1: *API for writing and reading compressed meshes.*

Current connectivity coders [TG98, GS98, Ros99] encode mesh connectivity with a practically constant 1 to 4 bits per vertex. They completely disregard the original element order of the input mesh by encoding the triangles in an order that is derived from systematically traversing the connectivity graph, and the vertex positions in the order they are first encountered. This way they basically eliminate the storage costs for specifying a particular ordering of the mesh elements. For planar triangulations, Poulalhon and Schaeffer [PS03] show how to entirely eliminate these costs by constructing a truly "canonical" ordering. Recent schemes further improve connectivity coding with mesh traversals that adapt to the topological [AD01] or geometrical [LAD02, KPRW05] regularity found in most meshes.

Although compression strategies based on systematic traversals can achieve the best possible connectivity bit-rates, they struggle with large meshes. Before compression can start they need to construct data structures that support the topological adjacency queries that are needed for traversing the connectivity graph. For meshes that contain hundreds of millions of triangles, such as the 3D scans of Michelangelo's statues [LPC*00], the construction of these temporary data structures is difficult given the limited main memory available on common PCs . Either the meshes must be cut into smaller pieces [HLK01] or external memory data structures [IG03] must be used. Both these approaches are I/O-inefficient and require a lot of temporary disk space.

Streaming meshes [IL05] provide interleaved access to vertices and triangles and "finalize" vertices that are no longer used by subsequent triangles. This information allows I/O-efficient parsing and processing of large meshes that can not be entirely loaded into memory as one can safely complete operations on finalized vertices and deallocate their associated data structure. One example of this type of I/O-efficient processing is streaming simplification [ILGS03]. Another example is streaming compression.

Streaming compression is fundamentally different from non-streaming compression as the compressor is no longer free to globally reorder the mesh. It has to compress the triangles more or less in the order they arrive—either in the exact order or within a small, user-defined delay of d triangles. How much compression is possible will now also depend on the input triangle order. This adversely affects compression rates since at least some ordering information has to be stored, but it also "enforces" I/O-efficiency: at any time only the active elements of the input stream (and possibly a few delayed elements) need to be kept in memory.

2.1. Streaming Meshes

A *streaming mesh* logically interleaves vertices and the triangles that reference them and provides explicit information about when vertices are "finalized" or "referenced for the last time". Vertices become *active* when they appear in the stream and cease to be active when they are finalized. The *width* is the maximal number of concurrently active vertices. It gives a lower bound on the memory requirements for processing a streaming mesh as we must minimally maintain all active vertices (e.g. in a hash table). The *span* is the maximal index difference between two active vertices and intuitively measures the longest duration a vertex remains active.

Streaming formats allow efficient storage of indexed connectivity because they only need a unique index for every active vertex and not for every vertex of the mesh. Finalization tells us that a vertex is not referenced by subsequent triangles—effective "freeing up" its index to be used again.

Relative Indexing references a vertex as the difference of the currently highest index to the absolute index of that vertex. Each index can then be stored with $\log_2(span)$ bits as the maximal index difference equals the span of the streaming mesh. Using the "current" span makes the total storage costs an integral of the logarithm of the span over the stream.

Dynamic Indexing references each of the w active vertices with an index that is within the range 0 to $w-1$. Each index can then be stored with $\log_2(width)$ bits as the maximal number of active vertices equals the width of the streaming mesh. Using the "current" width makes the storage cost an integral of the logarithm of the width over the stream.

To compare relative and dynamic indexing with standard indexing (see Table 1) we must add one bit per index needed to specify finalization. Relative indexing is at least as expensive as dynamic indexing, with equality being reached only if vertices are introduced and finalized in the same order. The storage costs for dynamic indexing are in general still superlinear. Even if we reorder the mesh to minimize its width, we are subject to the worst-case bound $\Theta(\sqrt{v})$ on the width of a mesh with v vertices [BYG96] so that the best we can guarantee is $\Theta(\log_2 \sqrt{v}) = \Theta(\log_2 v)$ bits per vertex.

3. Streaming Compression

Departing from the traditional approaches to mesh compression, which globally reorder mesh triangles based on a deterministic traversal of the connectivity graph, we describe a streaming compression scheme that can encode the triangles of the mesh in whatever order they are given to the compressor. Because such an encoding not only encodes the connectivity but also one of $t! \approx (2v)!$ possible triangle orderings, it requires in the worst case $\Theta(\log_2 v)$ bits per vertex. The expected coding costs, however, are much lower, especially if we allow the compressor to locally reorder triangles.

mesh (ordering)	bits per index abs	rel	dyn	largest index range rel (= span)	dyn (= width)
armadillo					
(vcompact)	17.4	17.4	16.1	172,715	51,951
(spectral)	17.4	10.6	9.3	4,405	638
(geometric)	17.4	11.0	10.0	3,796	1,042
(breadth)	17.4	10.2	10.1	1,197	1,129
(depth)	17.4	16.9	10.3	171,845	1,457
(rendering)	17.4	16.5	10.1	143,589	904
dragon					
(vcompact)	18.7	15.4	12.5	54,825	4,586
(spectral)	18.7	11.7	9.5	11,617	668
(geometric)	18.7	12.9	10.7	9,243	1,274
(breadth)	18.7	11.1	10.8	1,994	1,671
(depth)	18.7	18.4	12.8	436,834	8,587
(rendering)	18.7	16.2	9.8	126,152	944
lucy					
(vcompact)	23.7	23.3	18.4	13,500,197	255,446
(geometric)	23.7	13.6	12.8	20,362	4,985
david$_{1mm}$					
(vcompact)	24.7	23.8	14.8	15,821,388	26,383
(geometric)	24.7	13.5	13.1	36,421	8,919
st. matthew					
(vcompact)	27.5	25.2	15.2	29,189,836	31,931
(geometric)	27.5	15.2	14.8	157,920	33,207

Table 1: *Average number of bits per index for absolute, relative, and dynamic indexing in differently ordered streaming meshes. Also reported is the largest index range for relative and dynamic indexing, which correspond to span and width.*

What we gain for moderate losses in connectivity compression rates is the ability to encode meshes *while* they are produced and store them directly in compressed form to disk. We avoid having to wait for the entire mesh before compression can begin, having to build temporary data structures in the size of the mesh, and having to reorder the data on a global scale. Meshes can be compressed in increments of single vertices and triangles (see Figure 1), which makes compression essentially transparent to the user and practically independent of the mesh size, and therefore easier to integrate into the mesh processing pipeline.

The only requirement for compression is that the input is *pre-order* (i.e. each vertex is written before it is referenced by a triangle) and that finalization is *immediate* (i.e. each vertex is finalized with the last triangle that references it). Given such input, our compressor encodes the triangles in the exact order they are written and the vertices in the order they are first referenced. When vertices are written they are simply inserted into a hash table using their index as key. There they remain until the first triangle is written that references

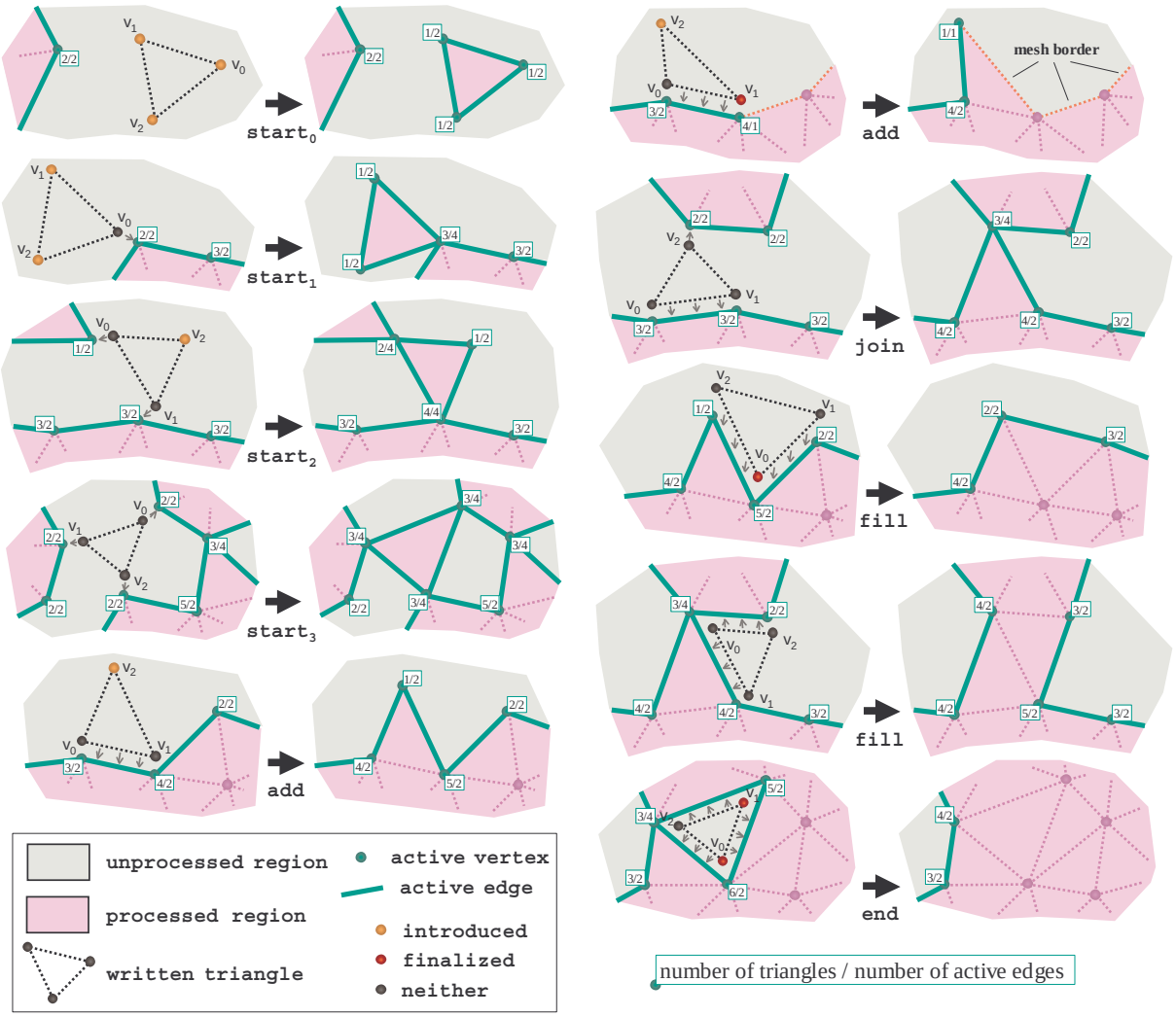

Figure 2: *The eight possible adjacency configurations between the written triangle and the active vertices and half-edges maintained by the compressor: a* ***start***$_x$ *triangle is not adjacent to any active half-edge, but may be adjacent to zero, one, two, or even three active vertices; an* ***add*** *triangle is only adjacent to one active half-edge, with the third vertex being newly introduced; for the similar* ***join*** *configuration this third vertex is already active; a* ***fill*** *triangle is adjacent to two half-edges and an* ***end*** *triangle is adjacent to three half-edges. The small boxes show the triangle count and number of active half-edges.*

them. This effectively delays vertices that were written "too early" and brings them into an ordering that is *compact* with respect to the triangle order. Hence, the compressed output mesh is always *vertex-compact* [IL05]. Meshes that are not pre-order or that do not immediately finalize vertices can be piped through a user-transparent on-the-fly converter.

3.1. Compression Details

The compressor maintains a set of *active vertices* and a set of *active half-edges*. A vertex becomes active when the first triangle that references it is written, and remains active until finalized by the last triangle that references it. Up to three half-edges can become active when a triangle is written. They remain active until their counterpart of opposite orientation appears or one of their incident vertices is finalized. Each active vertex stores a list of pointers to its incident half-edges. Each active half-edge stores pointers to its incident vertices and a copy of the position of its *across* vertex, which is the non-incident third vertex of the triangle that created this half-edge, that is required for predictive coding of positions.

Whenever a triangle is written the compressor checks which of the triangle's vertices and half-edges of opposite orientation are already active. The eight different configurations that can arise, namely *start*$_0$, *start*$_1$, *start*$_2$, *start*$_3$, *end*, *add*, *join*, and *fill*, are illustrated in Figure 2. The compressor encodes the current configuration with an arithmetic coder using four different symbols: START, ADD, JOIN,

FILL_END. Because $start_x$ configurations are infrequent for reasonably coherent orderings (especially after local reordering of triangles) only one symbol is used for all four, and x is compressed in a separate context. The *fill* and *end* configurations can be distinguished at the decoding end.

Next all the active vertices of the triangle (or none in a $start_0$ configuration) are referenced. If w is the current number of active vertices, then this can be done with dynamic indexing using $\log_2(w)$ bits per vertex. But because consecutive triangles often share vertices (especially after local reordering of triangles), we first check whether a vertex was used by the previous triangle and if so encode which of the three vertices it was. This often avoids the $\log_2(w)$ bits that are the most expensive part of our connectivity encoding.

For an *add*, *join*, *fill*, or *end* configuration the current triangle is also adjacent to one or more active half-edges. We can reference other active vertices using the list of half-edges maintained with the first referenced vertex. Since this list typically contains only one half-edge with the correct orientation, we often avoid storing any further information. Only vertex v_2 of a *join* configuration cannot be referenced this way and instead requires dynamic indexing, which makes a *join* the most expensive configuration to encode.

For an *add* configuration we predict the position of the newly introduced vertex with the parallelogram rule [TG98], using the two incident vertices and the *across* vertex of the adjacent active half-edge. For all $start_x$ configurations we predict the positions of newly introduced vertices as that of a known neighboring vertex. Since the first vertex of a $start_0$ configuration has no known neighbor, we simply use the most-recently compressed vertex position as the prediction. We compress the resulting corrective vectors with different arithmetic contexts depending on whether parallelogram, neighbor, or most-recent prediction was used.

Finalization information is specified for all three vertices of the written triangle with binary flags that can be efficiently compressed with context-sensitive arithmetic coding. The context is chosen based on the current number of triangles and active half-edges around this vertex. As most vertices are finalized when they are surrounded by a closed ring of triangles there is a strong correlation between the moment a vertex no longer has active half-edges and its finalization. Border vertices, which will still have one or two active half-edges tend to be surrounded by fewer triangles.

The vertices are maintained in two data structures: a hash table and a dynamic vector. The hash table is used to look up vertices by their index. A vertex is added to the hash table when it is written, it is looked up when a triangle that references it is written, and it is removed when it is finalized. The dynamic vector is used to explicitly reference active vertices (if necessary) with dynamic indices between 0 and $w-1$ that can then be stored with $\log_2(w)$ bits. A vertex is added to the dynamic vector when it is referenced for the first time, its index is looked up whenever the encoder needs to reference it explicitly, and it is removed when it is finalized.

```
bool write_vertex(float* v_pos) {
  Vertex* v = allocVertex(v_pos);
  hash->insert(v, v_count);
  v_count++;
}

bool write_triangle(int* t_idx, bool* t_final) {
  Vertex* v[3];
  for (i = 0; i < 3; i++) {
    v[i] = hash->get(t_idx[i]);
    if (v[i] == 0) return false; // must be pre-order
  }
  determine and compress configuration;
  rotate triangle so vertex v0 of Figure 2 is in v[0];
  if (START_x configuration) {
    compress which START_x it is;
    for (i = 0; i < 3; i++) {
      if (x- -) {         // v[i] is an old vertex
        if (v[i] was used by previous triangle) {
          compress which of its vertices is re-used;
        } else {
          index = dvector->get_index(v[i]);
          store dynamic index of v[i];
        }
      } else {            // v[i] is a new vertex
        dvector->add(v[i]);
        compress position of v[i];
      }
    }
    create three new half-edges;
  } else if (ADD or JOIN configuration) {
    if (v[0] or v[1] was used by previous triangle) {
      compress which of its vertices is is re-used;
    } else {
      index = dvector->get_index(v[0]);
      store dynamic index of v[0];
    }
    compress which half-edge of v[0] leads to v[1];
    if (ADD configuration) {
      dvector->add(v[2]);
      compress position of v[2];
    } else {
      index = dvector->get_index(v[2]);
      store dynamic index of v[2];
    }
    create two new and delete one old half-edge;
  } else if (FILL or END configuration) {
    if (v[0] or v[1] or v[2] was used by prev. triangle) {
      compress which of its vertices is re-used;
    } else {
      index = dvector->get_index(v[0]);
      store dynamic index of v[0];
    }
    compress which half-edge of v[0] leads to v[1];
    compress which half-edge of v[0] leads to v[2];
    create new and delete old half-edges;
  }
  for (i = 0; i < 3; i++) {
    compress (which context) if v[i] is finalized
    if (t_final[i]) {
      dvector->remove(v[i]);
      hash->erase(v[i]);
      delete all half-edges of v[i];
      deallocVertex(v[i]);
    }
  }
}
```

Figure 3: *An implementation of our streaming compression algorithm (without delay buffer) in C-like pseudocode.*

The dynamic vector implements constant-time insertion and deletion of vertices and constant-time lookup of dynamic indices by making new vertices the last entry and by replacing deleted entries with the currently last entry. This means that the indices with which active vertices can be referenced in the dynamic vector change over time, but they do so in a consistent manner at both encoder and decoder.

Reading the pseudo-code of Figure 3 may further clarify the compression algorithm that we have just described.

mesh (ordering)	configurations [%] s	a	j	f	e	use [%]	details for conn [bpv] fig	pre	dyn	adj	fin	totals [bpv] conn	geom	time [sec]	mem [MB]
armadillo															
(vcompact)	20	15	15	30	20	**10**	4.2	1.3	32.7	1.1	.00	**39.35**	22.24	1.4	9.3
(spectral)	.0	50	.9	48	.5	**49**	1.3	1.1	8.5	.04	.01	**11.04**	19.15	.8	.8
(geometric)	.7	49	4.3	42	4.3	**51**	2.0	2.0	9.2	.14	.01	**13.26**	19.16	.8	.8
(breadth)	.0	50	1.4	47	1.4	**97**	1.8	0.9	0.6	.01	.01	**3.27**	19.14	.7	.9
(depth)	.0	50	2.5	45	2.5	**97**	2.0	0.1	0.5	.03	.01	**2.66**	19.17	.7	1.0
(rendering)	.5	49	4.7	41	4.6	**91**	2.5	3.1	1.7	.14	.01	**7.45**	19.35	.7	.8
dragon															
(vcompact)	15	24	9.6	37	14	**50**	3.9	3.9	13.0	.83	.04	**21.62**	21.49	2.6	1.6
(spectral)	.4	50	1.5	47	1.5	**49**	1.8	1.2	8.8	.07	.04	**12.02**	22.04	1.8	.8
(geometric)	1.1	48	3.7	43	3.8	**57**	2.2	2.4	8.7	.14	.04	**13.45**	21.86	1.8	.9
(breadth)	.0	50	3.1	44	3.1	**94**	2.1	1.8	1.3	.05	.04	**5.23**	21.98	1.8	.9
(depth)	.0	50	5.1	40	5.1	**94**	2.3	0.5	1.4	.09	.04	**4.35**	21.96	1.9	1.8
(rendering)	.6	49	5.1	40	4.9	**90**	2.5	3.1	1.8	.15	.04	**7.62**	22.13	1.8	.8
lucy															
(vcompact)	1.6	47	8.1	34	8.7	**77**	2.5	2.2	8.6	.35	.00	**13.60**	14.70	77	37
(geometric)	.5	49	2.1	46	2.1	**53**	1.9	1.9	11.3	.07	.00	**15.18**	14.58	65	1.6
david$_{1mm}$															
(vcompact)	12	28	5.8	44	9.4	**66**	2.8	2.6	9.9	.60	.02	**15.94**	10.95	126	4.8
(geometric)	.8	49	2.0	47	2.0	**67**	1.9	2.8	8.0	.07	.02	**12.71**	11.63	131	2.4
st. matthew															
(vcompact)	11	31	6.2	44	8.5	**67**	2.7	2.4	10.0	.53	.02	**15.62**	8.22	865	5.2
(geometric)	.9	48	2.2	46	2.3	**69**	1.9	2.9	8.6	.08	.02	**13.60**	8.97	907	4.0

ooc-compressor [IG03]		
[bpv]	time	mem
conn	prepro.	disk
geom	compr.	main
lucy ooc		
1.88	19 min	0.9 GB
14.60	5 min	128 MB
david ooc		
1.79	36 min	1.7 GB
11.32	14 min	192 MB
st. matthew ooc		
1.84	7 hrs	11 GB
8.83	4 hrs	384 MB

Table 2: *For streaming compressing we report the percentages of **s**tart, **a**dd, **j**oin, **f**ill, and **e**nd configurations and of subsequent triangles that re-**use** vertices. We give itemized coding costs for triangle con**fig**uration, **pre**vious and **dyn**amic references, edge **adj**acency, and vertex **fin**alization. Total bit-rates for **conn**ectivity and **geom**etry (quantized at 16 bits) and both **time** and **mem**ory footprint for reading, compressing, and writing the meshes on a 1.1 GHz Dell Inspiron laptop are listed.*

3.2. Quantization without Bounding Box

Most data sets do not make use of full floating-point precision so that the lowest-order bits are noise and not actual data. For effective compression, floating-point positions are usually quantized onto a uniform grid. To support quantization for streaming meshes whose bounding box is not known in advance, we use a scheme that quantizes conservatively using a bounding box that is learned as the mesh streams by. The first two vertex positions are compressed without quantization and their distance gives the initial guess on the number of mantissa bits that need to be preserved to guarantee the user-requested precision. This maximum distance is updated with every compressed vertex position and will eventually match the extent of the actual bounding box. How long quantization is overly conservative depends on the order in which of the vertex positions are compressed.

This scheme is part of our current API and works reasonably well, but we still need to analyze and optimize compression speeds and bit-rates. Since conservative quantization encodes many positions with more precision than needed, thereby inflating compression rates, we want to use bounding box information if possible. For the results reported in this paper we assume that advance knowledge about the bounding box is available. Our streaming mesh writer also supports lossless floating-point compression [ILS05]. This is less efficient since the low-order bits of the mantissa typically contain incompressible noise. But providing this functionality makes it possible to use compression when quantization—for whatever reason—is not an option.

3.3. Results

Detailed performance measurements of our streaming compressor for meshes written in different triangle orderings are listed in Table 2. The *vcompact* meshes have the triangles in their original order, the *geometric* meshes have the triangles sorted along one coordinate axis, and the *spectral* meshes have the triangles in a spectral order that minimizes the width [IL05]. The *breadth* meshes have the triangles in an order derived from a breadth-first ordering of the vertices, the *depth* meshes have the triangles in depth-first order, and the *rendering* meshes have a triangle order created by the recursive cut algorithm of Bogomjakov and Gotsman [BG01].

The rates for connectivity compression are, as anticipated, much higher than those of schemes that globally reorder triangles, whereas the rates for geometry compression are similar. We need to encode many dynamic indices when the re-use of vertices among subsequent triangles is low, which results in poor compression rates. Re-use is low when triangles appear "randomly", as in some vcompact orderings, but also when they "hop around" the advancing front, as in the spectral and geometric orderings. Topological breadth- or depth-first sorts derive triangle orders from local adjacency of mesh elements, which give high percentages of vertex re-use and, therefore, the best compression rates.

The biggest advantage of our compressor are the high speed and the small memory footprint in which it can compress even the largest models. The only other method that can compress models as large as, for example, the "St. Matthew" statue is the ooc-compressor by Isenburg and Gumhold [IG03], for which we report compression rates and time and memory consumption in Table 2. On a 2.8 GHz Pentium IV processor, they spent 7 hours creating an 11 gigabyte data structure on disk before the actual compression, which then took another 4 hours and 384 megabytes of main memory. In contrast, running on a 1.1 GHz mobile Pentium III we complete compression after 15 minutes while using only 6 megabytes of main memory and no temporary disk space. Their 11 hour and 11 gigabyte effort pays off with state-of-the-art connectivity compression rates that make the connectivity size 20% of the geometry size, instead of 150%. But so far we have not reordered a single triangle.

4. Reordering in a Delay Buffer

If we allow the compressor to locally reorder triangles with a greedy strategy that increases the vertex re-use among subsequent triangles, we can significantly improve the compression rates. We give our compressor the option to store a user-specified number of triangles in a *delay buffer* from which it can choose any triangle for output. To globally preserve the stream order we place a strict constraint on the maximum delay with which a triangle can leave the buffer. Setting this delay to be the size of the buffer allows to efficiently implement buffering and delay control with a simple ring buffer.

Our streaming compressor uses a greedy strategy that picks the next triangle with the following priority order from the buffer: First are triangles that share two vertices with the previous triangle. Ties between candidates are broken by using the triangle with the "oldest" shared vertex (i.e. that has the lowest index). Second are triangles that share only one vertex but also finalize some vertex. Third are triangles that that share one vertex and have only one active vertex. Ties between candidates are broken by using the triangle with the "newest" active vertex (i.e. that has the highest index). Fourth are triangles that that share one vertex and have two active vertices (neither of which is finalized, or such triangles would come second). Distinguishing this case as the fourth priority disfavors expensive *join* operations. Fifth are triangles that share one vertex but have no other active vertex. Sixth is the oldest triangle in the delay buffer.

4.1. Results

We have implemented this greedy reordering scheme and report in Table 3 what effect different delay buffer sizes, ranging from 25 to 50,000 triangles, have on the compression rates. Probably the biggest surprise is the enormous improvement in connectivity compression we can get with a delay buffer as small as 25 triangles. For the "David" and "St. Matthew" models, which contain 56 and 372 million triangles, a maximal delay of 25 from the original triangle order improves the bit-rates from around 16 down to around

mesh	bit-rates for delay buffers of different size [bpv]							
(ordering)	none	25	100	500	1,000	5,000	10,000	50,000
armadillo								
(vcompact)	39.4	34.7	31.5	29.3	28.6	23.7	20.5	12.9
(spectral)	11.0	10.3	8.9	5.5	4.4	3.7	3.6	3.5
(geometric)	13.3	10.2	8.3	5.6	4.5	3.6	3.5	3.5
(breadth)	3.3	2.5	2.6	3.1	3.3	3.6	3.6	3.5
(depth)	2.7	2.7	2.9	3.4	3.8	3.6	3.6	3.5
(rendering)	7.5	6.0	4.9	4.2	4.0	3.7	3.7	3.6
dragon								
(vcompact)	21.6	15.2	12.1	11.2	11.1	9.5	7.4	4.8
(spectral)	12.0	11.0	9.5	6.3	5.3	4.4	4.3	4.3
(geometric)	13.5	9.7	8.3	6.8	6.1	4.7	4.5	4.4
(breadth)	5.2	3.7	3.7	3.8	4.0	4.6	4.5	4.4
(depth)	4.4	4.5	4.5	4.6	4.9	5.1	4.9	4.6
(rendering)	7.6	6.1	5.2	4.7	4.6	4.5	4.4	4.4
lucy								
(vcompact)	13.6	12.7	12.3	12.0	11.8	9.7	8.4	6.0
(geometric)	15.2	13.6	13.3	12.1	10.5	6.2	5.0	3.8
david$_{1mm}$								
(vcompact)	15.9	7.1	5.5	4.6	4.5	4.0	3.8	3.5
(geometric)	12.7	7.6	6.8	6.1	5.7	4.8	4.2	3.7
st. matthew								
(vcompact)	15.6	6.8	5.5	4.5	4.3	4.0	3.9	3.6
(geometric)	13.6	7.7	6.9	6.4	6.2	5.4	5.2	3.9

Table 3: *Bit-rates for streaming connectivity compression with delay buffers of different size. Respecting the maximal allowed delay, triangles are greedily chosen from the buffer to maximize vertex re-use among subsequent triangles.*

7 bits per vertex. The bit-rates of all mesh orderings seem to converge towards 4 to 5 bpv as the buffer size is increased to 50,000 triangles. For the *breadth* and *depth* orderings adding a larger delay buffer slightly worsens the bit-rates. Local greedy decisions in the buffer can destroy the overall regularity that such global topological orderings possess. Only for the completely random triangle orderings that are sometimes found in smaller models like the "Armadillo" or the popular "Stanford bunny", local reordering is not sufficient.

Delay buffer of 10,000 to 50,000 triangles constitute a comparatively minor amount of main memory for compressing models of several hundred million triangles. A delay buffer of 10,000 triangles, for example, increases the memory footprint by only 5 MB (not optimized yet) while slowing down compression speeds by a factor of two (not optimized yet). However, it gives connectivity compression rates that are within a factor of two of those of the ooc-compressor [IG03]. Memory usage and timings can easily be improved by integrating the delay buffer into the compressor. Currently the delay buffer is implemented as a filter through which the mesh is piped before being given to the compressor. This allows a modular implementation but requires a second hash table and other duplicate data structures.

5. Alternate Approaches to Streaming Compression

This paper has addressed a slightly different problem than previous works in mesh compression, in particular, in terms

of connectivity coding. While the connectivity of the mesh is compressed, a pre-existing ordering of the triangles has to be preserved, either exactly or within a certain tolerance (e.g. a maximal allowed delay). We have presented one possible solution to this problem that makes use of a greedy re-ordering technique. But given the large number of different schemes for non-streaming mesh compression, there surely are many alternate approaches for streaming compression yet to be investigated. We experimented with one other approach.

The described SMC compressor needs to encode at least one reference per triangle to an active vertex—either to a vertex of the previous triangle or with a dynamic index. In contrast to non-streaming schemes it lacks the determinism of choosing "on its own" where (i.e. adjacent to which half-edge) to encode the next triangle. Our experimental SMD compressor deterministically chooses where to encode the next triangle but corrects the choice with a special command if the respective triangle is not yet in the delay buffer.

Obviously there is no deterministic strategy that can avoid corrections altogether, unless we buffer *all* the triangles (which is exactly how standard compressors operate). Our SMD compressor implements a simple breadth-first strategy on the half-edges, which are kept in a traversal queue. At every step the compressor tries to compress the triangle adjacent to the next half-edge in the queue. If this triangle is not available, a *skip* command is encoded and the compressor encodes the oldest triangle in the delay buffer instead. Initial results indicate that if there are less than two percent of *skip* commands, the SMD compressor can beat the SMC compressor and give bit-rates nearly as good as those of the ooc-compressor [IG03]. However, for higher percentages of *skip* commands the compression rates can also be much worse.

6. Summary and Conclusion

We have described a streaming compression scheme that allows encoding meshes *on-the-fly* by operating on a partial representation of the connectivity that is created *and* deleted as the mesh is fed in increments of single triangle and vertices to the compressor. In contrast, all previous schemes [TG98, GS98, Ros99, LAD02, IG03, KPRW05] expect the entire mesh up front, and first construct temporary data structures that allow them to traverse the mesh connectivity at will before the actual compression can start.

The advantage of a streaming compressor grows with the size of the input mesh, as both construction and use of these temporary representations become increasingly cumbersome. For the 372 million triangle "St. Matthew" statue, the compressor by Isenburg and Gumhold [IG03] first spends 7 hours creating an 11 gigabyte data structure on disk before the actual compression starts, which then takes another 4 hours and uses 384 megabytes of main memory. In contrast, we complete compression in 15 minutes using only 6 megabytes of main memory and no temporary disk space.

When we preserve the exact triangle order our connectivity compression rates are considerably worse than those of Isenburg and Gumhold [IG03]. However, employing a small buffer within which we delay triangles to put them into an order that is more "compressible" allows us to significantly improve compression. While streaming connectivity compression will in general not be able to rival rates achieved by non-streaming approaches, streaming I/O makes compression a more useful tool in a typical mesh processing pipeline because it remains transparent to the user.

References

[AD01] ALLIEZ P., DESBRUN M.: Valence-driven connectivity encoding for 3D meshes. In *Eurographics'01 Proceedings* (2001), pp. 480–489. 2

[BG01] BOGOMJAKOV A., GOTSMAN C.: Universal rendering sequences for transparent vertex caching of progressive meshes. In *Graphics Interface'01 Proceedings* (2001), pp. 81–90. 6

[BYG96] BAR-YEHUDA R., GOTSMAN C.: Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics 15*, 2 (1996), 141–152. 3

[Dee95] DEERING M.: Geometry compression. In *SIGGRAPH 95 Conference Proceesings* (1995), pp. 13–20. 1

[GS98] GUMHOLD S., STRASSER W.: Real time compression of triangle mesh connectivity. In *SIGGRAPH'98 Conference Proceesings* (1998), pp. 133–140. 1, 2, 8

[HLK01] HO J., LEE K., KRIEGMAN D.: Compressing large polygonal models. In *Visualization 2001*, pp. 357–362. 1, 2

[IG03] ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. In *SIGGRAPH 2003 Proceedings* (2003), pp. 935–942. 1, 2, 6, 7, 8

[IL05] ISENBURG M., LINDSTROM P.: *Streaming Meshes*. Technical Report, UCRL-CONF-201992, LLNL, 2005. 2, 4, 6

[ILGS03] ISENBURG M., LINDSTROM P., GUMHOLD S., SNOEYINK J.: Large mesh simplification using processing sequences. In *Visualization'03 Proc.* (2003), pp. 465–472. 2

[ILS05] ISENBURG M., LINDSTROM P., SNOEYINK J.: Lossless compression of predicted floating-point geometry. *Computer-Aided Design 37*, 8 (2005), 869–877. 6

[KPRW05] KÄLBERER F., POLTHIER K., REITEBUCH U., WARDETZKY M.: *FreeLence - Coding with free valences*. to appear in *Eurographics'05 Proceedings* (2005). 1, 2, 8

[LAD02] LEE H., ALLIEZ P., DESBRUN M.: Angle-analyzer: A triangle-quad mesh codec. In *Eurographics'02 Proceedings* (2002), pp. 198–205. 1, 2, 8

[LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: The Digital Michelangelo Project. In *SIGGRAPH'00*, pp. 131–144. 1, 2

[PS03] POULALHON D., SCHAEFFER G.: Optimal coding and sampling of triangulations. In *30th Intern. Colloq. on Automata, Languages and Program. (ICAZLP)* (2003), pp. 1080–1094. 2

[Ros99] ROSSIGNAC J.: Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 5*, 1 (1999), 47–61. 1, 2, 8

[TG98] TOUMA C., GOTSMAN C.: Triangle mesh compression. In *Graphics Interface'98 Proceedings* (1998), pp. 26–34. 1, 2, 5, 8